

# 目录

---

实验任务 1 .....	4
实验任务 2 .....	8
实验任务 3 .....	13
实验任务 4 .....	18
1.结合代码分析虚拟页内存分配的三步过程和虚拟页内存释放。 .....	18
2.构造测试例子来分析虚拟页内存管理的实现是否存在 bug。如果存在，则尝试 修复并再次测试。否则，结合测例简要分析虚拟页内存管理的实现的正确性。 .....	25



# 本科生实验报告

实验课程：\_\_\_\_\_操作系统原理实验\_\_\_\_\_

实验名称：\_\_\_\_\_内存管理\_\_\_\_\_

专业名称：\_\_\_\_\_计算机科学与技术\_\_\_\_\_

学生姓名：\_\_\_\_\_张玉瑶\_\_\_\_\_

学生学号：\_\_\_\_\_23336316\_\_\_\_\_

实验地点：\_\_\_\_\_实验楼 B203\_\_\_\_\_

实验成绩：\_\_\_\_\_

报告时间：\_\_\_\_\_2025 年 5 月 18 日\_\_\_\_\_

## Section 1 实验概述

- 实验任务 1: 复现参考代码，实现二级分页机制，并能够在虚拟地址空间中进行内存管理，包括内存的申请和释放等，截图并给出过程解释。
- 实验任务 2: 参照理论课上的学习的物理内存分配算法如 first-fit, best-fit 等实现动态分区算法等，或者自行提出自己的算法。
- 实验任务 3: 参照理论课上虚拟内存管理的页面置换算法如 FIFO、LRU 等，实现页面置换，也可以提出自己的算法。
- 实验任务 4: 复现“虚拟页内存管理”一节的代码，完成如下要求：
  1. 结合代码分析虚拟页内存分配的三步过程和虚拟页内存释放。
  2. 构造测试例子来分析虚拟页内存管理的实现是否存在 bug。如果存在，则尝试修复并再次测试。否则，结合测例简要分析虚拟页内存管理的实现的正确性。

## Section 2 实验步骤与实验结果

### ----- 实验任务 1 -----

- 任务要求: 复现参考代码，实现二级分页机制，并能够在虚拟机地址空间中进行内存管理，包括内存的申请和释放等，截图并给出过程解释。
- 思路分析: 本实验的核心在于实现并启动二级分页机制，规划页目录表和页表的位置，并通过 CR3 和 CR0 控制寄存器完成分页开启。在建立 0~1MB 线性地址到物理地址的恒等映射时，需要正确设置页目录项和页表项，包括地址对齐和访问权限位。为了实现虚拟地址空间中的内存管理，我们完成了基本的内存页申请和释放功能，并验证了内存首地址的变化，说明分页机制成功运行，尽管在空指针释放处理上仍需完善。
- 实验步骤:
  1. 规划好页目录表和页表在内存中的位置，然后初始化。

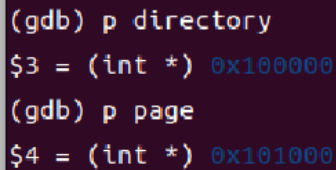
页目录表和页表是需要在内存中特意地划分出位置来存放的。所以，我们需要在内存中指定页目录表和页表存放的位置。同时，页目录表和页表的物理地址必须是 4KB 的整数倍，也就是低 12 位为 0。

规定了页目录表的位置后,我们根据线性地址空间的大小来确定需要分配的页表的数量和位置,不必一开始就分配完 1024 个页表给页目录表。规划好了页目录表的位置后,我们向页目录表中写入页表对应的页目录项。

```
#define PAGE_DIRECTORY 0x100000
```

```
// 页目录表指针
int *directory = (int *)PAGE_DIRECTORY;
// 线性地址 0~4MB 对应的页表
int *page = (int *)(PAGE_DIRECTORY + PAGE_SIZE);
```

我们可以看到 directory 指向了 1MB 的位置,说明页目录表放在了 1MB 处。page 指向的位置说明页目录大小为 4KB。



```
(gdb) p directory
$3 = (int *) 0x1000000
(gdb) p page
$4 = (int *) 0x1010000
```

初始页目录项。

```
// 初始化页目录表
memset(directory, 0, PAGE_SIZE);
// 初始化线性地址 0~4MB 对应的页表
memset(page, 0, PAGE_SIZE);
```

规定了页目录表的位置后,我们根据线性地址空间的大小来确定需要分配的页表的数量和位置,不必一开始就分配完 1024 个页表给页目录表。

为了访问方便,对于 0~1MB 的内存区域我们建立的是虚拟地址到物理地址的恒等映射,也就是说,虚拟地址和物理地址相同。这个时候,我们就要设置相应的页目录项和页表项。

首先,虚拟地址 0~1MB 的二进制表示是 0x00000000~0x000fffff,其 31-22 位均为 0,对应第 0 个页目录项。因此我们只需要初始化第 0 个页目录项和其对应的页表即可。第 0 个页目录项被放在页目录表之后,地址是 PAGE\_DIRECTORY + PAGE\_SIZE。然后我们取 21~12 位,范围从 0x000~0xfff,涉及 256 个页表项。由于我们希望线性地址经过翻译后的物理地址依然和线性地址相同。因此,这

256 个页表项分别指向物理页的第 0 页，第 1 页和第 255 页。

除了设置页表项对应的物理页地址和固定为 0 的位外，我们设置 U/S, R/W 和 P 位为 1。

```
int address = 0;
// 将线性地址 0~1MB 恒等映射到物理地址 0~1MB
for (int i = 0; i < 256; ++i)
{
    // U/S = 1, R/W = 1, P = 1
    page[i] = address | 0x7;
    address += PAGE_SIZE;
}
```



```
(gdb) p/x page[0]
$8 = 0x7
(gdb) p/x page[1]
$9 = 0x1007
(gdb) p/x page[2]
$10 = 0x2007
```

可以观察到所有页表项的 31~12 位表示页表的物理地址，已经将线性地址 0~1MB 恒等映射到物理地址 0~1MB。同时与 7 或操作，可以让每一个页表项的最后三位都为一，实现  $U/S = 1$ ,  $R/W = 1$ ,  $P = 1$ 。

初始化页目录项。由于我们的 0~1MB 的线性地址对应于第 0 个页目录项，我们用刚刚放入了 256 个页表项的页表作为第 0 个页目录项指向的页表。同样地，我们设置 U/S, R/W 和 P 位为 1。

```
// 初始化页目录项

// 0~1MB
directory[0] = ((int)page) | 0x07;
// 3GB 的内核空间
directory[768] = directory[0];
// 最后一个页目录项指向页目录表
```

```
directory[1023] = ((int)directory) | 0x7;

// 初始化 cr3, cr0, 开启分页机制
asm_init_page_reg(directory);
```

我们可以观察到第 768 个页目录项和第 0 个页表项相同、设置最后一个页目录项指向页目录表，这是用于构造页目录项和页表项的虚拟地址而服务的。

```
(gdb) p/x directory[0]
$17 = 0x101007
(gdb) p/x directory[768]
$18 = 0x101007
(gdb) p/x directory[1023]
$19 = 0x100007
```

## 2. 将页目录表的地址写入 cr3。

cr3 寄存器保存的是页目录表的地址，使得 CPU 的 MMU 能够找到页目录表的地址，然后自动地将线性地址转换成物理地址。我们在建立页目录表和页表后，需要将页目录表地址放到 CPU 所约定的地方，而这个地方是 cr3。cr3 又被称为页目录基址寄存器 PDBR。

```
(gdb) p directory
$3 = (int *) 0x100000
(gdb) p page
$4 = (int *) 0x101000
(gdb) p/x $cr3
$7 = 0x100000
```

## 3. 将 cr0 的 PG 位置 1。

启动分页机制的开关是将控制寄存器 cr0 的 PG 位置 1，PG 位是 cr0 寄存器的第 31 位，PG 位为 1 后便进入了内存分页运行机制。

```
(gdb) p/x $cr0
$8 = 0x80000011
```

## 4. 实现内存的申请和释放。

对 p1 先进行分配，后全部释放。

```

char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
//char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
//char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);

printf("%x\n", p1);

memoryManager.releasePages(AddressPoolType::KERNEL, (int)p1, 100);
//p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);

printf("%x\n", p1);

```

输出如下。第一个输出是 p1 得到分配的内存首地址。第二个是释放了 p1 全部内存后的输出。这里应该是有 bug，应该让 p1 输出 nullptr 才对。这个在 assignment4 处再做修改。

```

C0100000
C0100000

```

- 实验结果展示：通过执行前述代码，可得下图结果。

```

Booting from Hard Disk...
open page mechanism
total memory: 133038080 bytes ( 126 MB )
kernel pool
(   start address: 0x200000
(   total pages: 15984 ( 62 MB )
(   bitmap start address: 0x10000
user pool
(   start address: 0x4070000
(   total pages: 15984 ( 62 MB )
(   bit map start address: 0x107CE
kernel virtual pool
(   start address: 0xC0100000
(   total pages: 15984 ( 62 MB )
(   bit map start address: 0x10F9C
C0100000
C0100000

```

## ----- 实验任务 2 -----

- 任务要求：参照理论课上的学习的物理内存分配算法如 first-fit, best-fit 等实现动态分区算法等，或者自行提出自己的算法。我将实现 best-fit 算法并举例验证。
- 思路分析：本实验实现了基于 Best-Fit 策略的动态分区物理内存分配算法。通过对原先的 First-Fit 策略进行替换，优化了内存利用率，使得内存块分配更加贴合实际需求。具体方法是在每次分配时遍历整个位图，寻找最小但足

够的连续空闲区域，并优先使用它。通过加入分配和释放的可视化输出，可以清晰观察算法选择的内存地址变化，最终验证了 Best-Fit 算法能够有效分配最合适的内存区域，提高了碎片利用率。

- 实验步骤：

修改原有的分配算法。原有算法为 first-bit，现在修改为 best-fit。

1.现在最顶层的 MemoryManager 中修改分配函数，仅添加打印内容显示分配情况，易于观测。

```
int MemoryManager::allocatePhysicalPages(enum AddressPoolType type,
const int count)
{
    int start = -1;
    //string type_of=to_string(type);
    if (type == AddressPoolType::KERNEL)
    {
        start = kernelPhysical.allocate(count);
    }
    else if (type == AddressPoolType::USER)
    {
        start = userPhysical.allocate(count);
    }

    if(start!=-1){
        printf("    allocate from 0x%x to 0x%x, size: %d\n",start,start+count*PAGE_SIZE-1,count);
    }
    return (start == -1) ? 0 : start;
}
```

2.再在最底层的 BitMap 中修改分配函数，每次调用分配函数则寻找符合要求的最小连续页，满足 best-fit 算法。

```
int BitMap::allocate(const int count)
{
    if (count == 0)
        return -1;

    int index, empty, start;
    int best_start=-1,best_size=INT_MAX;

    index = 0;
    while (index < length)
    {
        // 越过已经分配的资源
        while (index < length && get(index))
            ++index;

        // 不存在连续的 count 个资源
        if (index == length)
```



```

        return -1;

// 找到 1 个未分配的资源
// 检查是否存在从 index 开始的连续 count 个资源
//empty = 0;

int current_size=0;

start = index;
while ((index < length) && (!get(index)) )
{
    ++current_size;
    ++index;
}

if(current_size>=count && current_size<best_size){
    best_start=start;
    best_size=current_size;
}

// 存在连续的 count 个资源

if(best_start!=1){
    for(int i=0;i<count;i++){
        set(best_start+i,true);
    }
    return best_start;
}
return -1;
}

return -1;
}

```

3.修改释放函数。增加输出从何处释放到何处，共多少页，方便观察。

```

void MemoryManager::releasePhysicalPages(enum AddressPoolType type,
const int paddr, const int count)
{
    if (type == AddressPoolType::KERNEL)
    {
        kernelPhysical.release(paddr, count);
    }
    else if (type == AddressPoolType::USER)
    {
        userPhysical.release(paddr, count);
    }
    //string type_of=to_string(type);
    printf("release from 0x%x to 0x%x, size: %d\n",paddr,paddr+count*PAGE_SIZE-1,count);
}

```

4.在 setup.cpp 中修改中断处理函数。添加标黄内容。

```
extern "C" void setup_kernel()
{
    // 中断管理器
    interruptManager.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void
*)asm_time_interrupt_handler);

    // 输出管理器
    stdio.initialize();

    // 进程/线程管理器
    programManager.initialize();

    // 内存管理器
    memoryManager.openPageMechanism();
    memoryManager.initialize();

    int first=memoryManager.allocatePhysicalPages(USER,100);
    int second=memoryManager.allocatePhysicalPages(USER,100);
    int third=memoryManager.allocatePhysicalPages(USER,100);
    int fourth=memoryManager.allocatePhysicalPages(USER,10);
    int fifth=memoryManager.allocatePhysicalPages(USER,100);

    memoryManager.releasePhysicalPages(USER,second,100);
    memoryManager.releasePhysicalPages(USER,fourth,10);

    int sixth=memoryManager.allocatePhysicalPages(USER,5);
    //int seventh=memoryManager.allocatePhysicalPages(USER,10);

    // 创建第一个线程
    int pid = programManager.executeThread(first_thread, nullptr, "first
thread", 1);
    if (pid == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }

    ListItem *item = programManager.readyPrograms.front();
    PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
    firstThread->status = RUNNING;
    programManager.readyPrograms.pop_front();
    programManager.running = firstThread;
    asm_switch_thread(0, firstThread);

    asm_halt();
}
```

我们先利用内存管理器 memoryMenager 打开分页机制并初始化。

第二步，我们先在用户地址池分配五次内存，分配的页数分别是 100, 100,

100, 10, 100。在下图中我们可以看到连续输出了五次，显示了每一次分配的内存起始地址到终止地址和分配的页数。不同次分配成功！

```
int first=memoryManager.allocatePhysicalPages(USER,100);  
int second=memoryManager.allocatePhysicalPages(USER,100);  
int third=memoryManager.allocatePhysicalPages(USER,100);  
int fourth=memoryManager.allocatePhysicalPages(USER,10);  
int fifth=memoryManager.allocatePhysicalPages(USER,100);
```

```
allocate from 0x4070000 to 0x40D4000, size: 100 pages  
allocate from 0x40D4000 to 0x4138000, size: 100 pages  
allocate from 0x4138000 to 0x419C000, size: 100 pages  
allocate from 0x419C000 to 0x41A6000, size: 10 pages  
allocate from 0x41A6000 to 0x420A000, size: 100 pages
```

第三步，我们释放掉第二次和第三次分配到的内存。

```
memoryManager.releasePhysicalPages(USER,second,100);  
memoryManager.releasePhysicalPages(USER,fourth,10);
```

```
release from 0x40D4000 to 0x4137FFF, size: 100 pages  
release from 0x419C000 to 0x41A5FFF, size: 10 pages
```

此时我们有三段空闲内存，分别是刚刚释放掉的 0x40D4000 到 0x4137FFF 一共 100 页，0x419C000 到 0x41A5FFF 一共 10 页，和 0x4209FFF 之后非常大的、远大于 100 页的剩余内存。

第四步，我们再分配一次大小仅为 10 页的内存。如果按照 first-fit 算法，当算法搜寻到 0x40D4000 到 0x4137FFF 这一片足够大小的空间时就会分配。而按照 best-fit 算法，算法会搜寻最小的合适的空闲内存分配，于是算法理应找到 0x419C000 到 0x41A5FFF 这一片只有 10 页的空间找前 5 页进行分配，以下进行验证。

```
int sixth=memoryManager.allocatePhysicalPages(USER,5);
```

最后输出如下。

```
allocate from 0x419C000 to 0x41A1000, size: 5 pages
```

可以发现算法确实寻找的是最小满足需求的地方进行分配。我们实现了 best-fit 算法！

- 实验结果展示：

```

Booting from Hard Disk...
open page mechanism
total memory: 133038080 bytes ( 126 MB )
kernel pool
    start address: 0x200000
    total pages: 15984 ( 62 MB )
    bitmap start address: 0x10000
user pool
    start address: 0x4070000
    total pages: 15984 ( 62 MB )
    bit map start address: 0x107CE
    allocate from 0x4070000 to 0x40D4000, size: 100 pages
    allocate from 0x40D4000 to 0x4138000, size: 100 pages
    allocate from 0x4138000 to 0x419C000, size: 100 pages
    allocate from 0x419C000 to 0x41A6000, size: 10 pages
    allocate from 0x41A6000 to 0x420A000, size: 100 pages
    release from 0x40D4000 to 0x4137FFF, size: 100 pages
    release from 0x419C000 to 0x41A5FFF, size: 10 pages
    allocate from 0x419C000 to 0x41A1000, size: 5 pages

```

### ----- 实验任务 3 -----

- 任务要求：参照理论课上虚拟内存管理的页面置换算法如 FIFO、LRU 等，实现页面置换，也可以提出自己的算法。在这里我实现 FIFO 算法。
- 思路分析：利用链表实现先进先出的逻辑。设计结点记录每一次分配虚拟内存的起始地址和长度。当内存足够时，将结点加入链表的尾部。当内存不够时，操作头结点：当头结点记录的长度大于本次分配的长度，修改头结点的起始地址和长度；当头结点记录的长度恰好等于本次分配的长度，更新头结点；当头结点记录的长度小于本次分配的长度，循环判断直到找到新的头结点。分配得到内存后再在尾结点后加入新节点，并更新尾结点。
- 实验步骤：

1.定义结点结构体。一个结点用来记录一次分配，start 记录该次分配的首地址，count 记录该次分配的页数。

```

struct node{
    int start;//记录每一次分配的起始地址
    int count;//每次的分配长度
    node *next=nullptr;
    node(){}
    node(int start,int count):start(start),count(count){}
};

```

2.在 MemoryManager 内加入如下标注成员。

```

class MemoryManager
{

```

```

public:
    // 可管理的内存容量
    int totalMemory;
    // 内核物理地址池
    .....
    node a[100];
    node *kerhead;//指向内核 fifo 队列的头节点
    node *kertail;//指向内核 fifo 队列的尾节点

```

因为禁用了 new，所以用 node a[100]记录每一次分配的信息。kerhead 指向了链表的头结点用于“释放”，kertail 指向链表的尾结点用于“分配”。

在 initialize()中初始化指针。利用全局变量 global\_tmpp 初始化指针，避免临时变量的销毁令指针信息出错。

```

int num=0;
node global_tmpp=node(-1,-1);

void MemoryManager::initialize()
{
    this->totalMemory = 0;
    this->totalMemory = getTotalMemory();
    global_tmpp=node(-1,-1);
    this->kerhead=&global_tmpp;//头节点出，永远指向第一个节点
    this->kertail=kerhead;//尾节点进，每次分配指向新节点

    this-> a[100];
}

```

3. 在 MemoryManager 加入函数 releaseHead，用于手动实现页的释放。

```

public:
    MemoryManager();
    .....
    void releaseHead(enum AddressPoolType type,int count);

```

4.函数的实现。最主要的改动有两个函数，一是原有的函数 allocateVirtualPages，而是刚刚添加的函数 releaseHead。

1) 实现 **releaseHead**。

因为初始化 kerhead 的时候让它指向了结点 global\_tmpp，所以需要先跳过该节点。

将 kerhead->start 的值赋给 start。因为链表是顺序记录，所以 kerhead 永远指向当前链表最先分配的一次信息。我们准备以 kerhead 作为首地址开始释放并重新分配。

记录本次分配需要的长度 count 并防止给 c。做循环判断。当 c<this->kerhead->count 意味着头节点的大小足够分配，我们只需要改头结点的

start 和 count 即可。当 `c==this->kerhead->count` 意味着头结点的大小刚好足够，只需要让 `kerhead` 指向下一个结点。当 `c>this->kerhead->count` 意味着第一个结点的大小不足够，这时候需要继续判断之后的结点加在一起是否足够重新分配。

最后我们拿到需要释放的内存的首地址和长度、利用原有的函数 `releasePages` 进行释放。

```
void MemoryManager::releaseHead(enum AddressPoolType type, const int count){
    int start=-1;
    if(!this->kerhead) return;
    if(type==AddressPoolType::KERNEL){//内核

        if(this->kerhead->count==-1 || this->kerhead->count==0){
            this->kerhead=this->kerhead->next;//掠过初始化的头节
        }

        start=this->kerhead->start;//返回首地址

        int c=count;
        while(true){
            if(c<this->kerhead->count){//如果头节点的大小足够分配，改头地址
                this->kerhead->start+=count*4096;
                this->kerhead->count-=count;
                break;
            }else if(c==this->kerhead->count){//头节点大小刚好足够
                this->kerhead=this->kerhead->next;
                break;
            }
            else{//头节点大小不足够 count>virhead->count
                c-=this->kerhead->count;
                this->kerhead=this->kerhead->next;
            }
        }
    }
    releasePages(type, start, count);
}
```

2) 在 `allocateVirtualPages` 加入链表的实现逻辑。

```
int MemoryManager::allocateVirtualPages(enum AddressPoolType type, const int count)
{
    int start = -1;

    if (type == AddressPoolType::KERNEL)
    {
        start = kernelVirtual.allocate(count);
    }

    if(start==-1 && type == AddressPoolType::KERNEL){//系统没有分配成功，
```

```

说明空间不够了，要算法启动
    //从头节点开始释放
    releaseHead(type,count);
    start=kernelVirtual.allocate(count);//释放空间后再次尝试分配
}

//分配成功了
//从尾节点开始加入
this->a[num].start=start;
this->a[num].count=count;
node *cur=&a[num];
num++;

this->kertail->next=cur;
this->kertail=kertail->next;

    printf("    from    0x%x    to    0x%x,    page    size=%d\n",start,start+count*4096,count);

    return (start == -1) ? 0 : start;
}

```

在这个函数中加入标黄的部分。在原有的函数中 `start` 得到的是分配成功后的连续虚拟页的首地址。如果分配失败，`start` 将赋值为-1。当分配失败时说明内存不够分配了，这时我们应该开始手动释放内存。

进入 `releaseHead` 后我们修改了先进先出链表的头结点并且释放了最先进入的一部分内存。这时候可以再次用语句 `start = kernelVirtual.allocate(count)` 重新分配，这次可以分配成功了。

因为又一次分配了链表，所以我们需要记录这一次分配。我们利用成员 `node a[100]` 来存储临时的信息，目的是让尾结点指向新创建的、记录了新分配的首地址和长度的结点，并让尾结点更新为新结点。

我们打印本次分配的信息，输出起始地址到终止地址和页数，方便我们观察。

## 5.开始测试。

```

void first_thread(void *arg)
{
    char *p1 =(char *)memoryManager.allocatePages(KERNEL, 10000);
    from 0xC0100000 to 0xC2810000, page size=10000

    char *p2 = (char *)memoryManager.allocatePages(KERNEL, 5000);
    from 0xC2810000 to 0xC3B98000, page size=5000

    char *p3 = (char *)memoryManager.allocatePages(KERNEL, 1000);//从头分

```

```

from 0xC0100000 to 0xC04E8000, page size=1000
char *p4 = (char *)memoryManager.allocatePages(KERNEL, 2000);
from 0xC04E8000 to 0xC0CB8000, page size=2000
char *p5 = (char *)memoryManager.allocatePages(KERNEL, 12000);
from 0xC0CB8000 to 0xC3B98000, page size=12000
char *p6 = (char *)memoryManager.allocatePages(KERNEL, 3000); // 从头分
from 0xC0100000 to 0xC0CB8000, page size=3000
char *p7 = (char *)memoryManager.allocatePages(KERNEL, 2000);
from 0xC0CB8000 to 0xC1488000, page size=2000
printf("\n%x %x %x %x %x %x %x \n", p1, p2, p3, p4, p5, p6, p7);
C0100000 C2810000 C0100000 C04E8000 C0CB8000 C0100000 C0CB8000
asm_halt();
}

```

我们在 `first_thread` 中连续分配七次。由于虚拟内存总共有 15984 页。按照我们分配的页数，在第三次分配的前只剩下 984 页，不能够满足 1000 页的需求。这时候就会根据 FIFO 的顺序，操作头结点释放最开始分配的内存。于是我们可以发现第三次分配的首地址和第一次分配的地址是一样的，都是虚拟内存开始的地址 `c0100000`。后面的 `p4, p5` 都是跟在 `p3` 后释放先前分配的内存再进行分配。`p6` 和 `p3` 同理。

自此，我们成功实现了页面置换的 FIFO 算法！

- 实验结果展示：



```
QEMU
Machine View
Booting from Hard Disk...
open page mechanism
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0x10000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x10F9C
  from 0xC0100000 to 0xC2810000, page size=10000
  from 0xC2810000 to 0xC3B98000, page size=5000
  from 0xC0100000 to 0xC04E8000, page size=1000
  from 0xC04E8000 to 0xC0CB8000, page size=2000
  from 0xC0CB8000 to 0xC3B98000, page size=12000
  from 0xC0100000 to 0xC0CB8000, page size=3000
  from 0xC0CB8000 to 0xC1488000, page size=2000
C0100000 C2810000 C0100000 C04E8000 C0CB8000 C0100000 C0CB8000
```

#### ----- 实验任务 4 -----

- 任务要求：复现“虚拟页内存管理”一节的代码，完成如下要求。
  - 1.结合代码分析虚拟页内存分配的三步过程和虚拟页内存释放。
  - 2.构造测试例子来分析虚拟页内存管理的实现是否存在 bug。如果存在，则尝试修复并再次测试。否则，结合测例简要分析虚拟页内存管理的实现的正确性。
- 思路分析：对于第一个任务无需细讲。第二个任务主要是给函数 `releasePages` 加一个返回值和参数 `beforeSize`，以此判断比较并且更新地址。
- 实验步骤：
  - 1.结合代码分析虚拟页内存分配的三步过程和虚拟页内存释放。

我们在 `allocatePages` 中实现页内存的分配。

1) 从虚拟地址池中分配若干连续的虚拟页。

```
// 第一步：从虚拟地址池中分配若干虚拟页
int virtualAddress = allocateVirtualPages(type, count);
if (!virtualAddress)
{
    return 0;
}
```

```
bool flag;
int physicalPageAddress;
int vaddress = virtualAddress;
```

我们进行 3 次分配进行测试。进入 p1 的分配函数。

```
char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
```

虚拟页的分配通过函数 allocateVirtualPages 来实现。我们只在内核虚拟池中分配内存。

```
int allocateVirtualPages(enum AddressPoolType type, const int count)
{
    int start = -1;

    if (type == AddressPoolType::KERNEL)
    {
        start = kernelVirtual.allocate(count);
    }

    return (start == -1) ? 0 : start;
}
```

利用 gdb 进行调试。进入函数 allocateVirtualPages。

```
// 第一步：从虚拟地址池中分配若干虚拟页
int virtualAddress = allocateVirtualPages(type, count);
if (!virtualAddress)
```

```
int MemoryManager::allocateVirtualPages(enum AddressPoolType type, const int count)
{
    int start = -1;

    if (type == AddressPoolType::KERNEL)
    {
        start = kernelVirtual.allocate(count);
    }
}
```

经过打印得到了第一次分配的虚拟地址为 0xc0100000。

```
(gdb) p/x start
$1 = 0xffffffff
(gdb) next
(gdb) p/x start
$2 = 0xc0100000
```

kernel virtual pool  
start address: 0xC0100000

可以看到经过分配，start 得到了分配内存的首地址。为内核虚拟池的起始页。  
这一次虚拟内存的分配从 0xc0100000 开始，是连续的 100 页。

2) 对每一个虚拟页，从物理地址池中分配 1 页。

```
// 依次为每一个虚拟页指定物理页
for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
{
    flag = false;
    // 第二步：从物理地址池中分配一个物理页
    physicalPageAddress = allocatePhysicalPages(type, 1);
```

在这里我就进行 1 次循环，只对虚拟地址 0xc0100000 进行物理页的分配和映射。

```
// 依次为每一个虚拟页指定物理页
for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
{
    flag = false;
    // 第二步：从物理地址池中分配一个物理页
    physicalPageAddress = allocatePhysicalPages(type, 1);
    if (physicalPageAddress)
```

进入分配物理页的函数。

```
// 第二步：从物理地址池中分配一个物理页
physicalPageAddress = allocatePhysicalPages(type, 1);
```

start 得到的是分配的物理页的地址，仅一页。

```
int MemoryManager::allocatePhysicalPages(enum AddressPoolType type, const int count)
{
    int start = -1;

    if (type == AddressPoolType::KERNEL)
    {
        start = kernelPhysical.allocate(count);
    }
}
```

得到的物理地址为 0x200000; 可以发现内核的物理池首地址也是 0x200000。我们为虚拟页 0xc0100000 分配了物理页 0x200000. 之后我们要建立他们之间的映射关系。

```
(gdb) p/x start
$4 = 0x200000
```

```
kernel pool
start address: 0x200000
```

start 从 allocatePhysicalPages 中返回后赋值给了 physicalPageAddress。

```
// 依次为每一个虚拟页指定物理页
for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
{
    flag = false;
    // 第二步: 从物理地址池中分配一个物理页
    physicalPageAddress = allocatePhysicalPages(type, 1);
    if (physicalPageAddress)
    {
        flag = true;
    }
}
```

```
(gdb) p/x physicalPageAddress
$5 = 0x200000
```

接下来要进入第三步, 为虚拟页建立页目录项和页表项。

```
if (physicalPageAddress)
{
    //printf("allocate physical page 0x%x\n", physicalPageAddress);

    // 第三步: 为虚拟页建立页目录项和页表项, 使虚拟页内的地址经过分页机制
    flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
}
```

3) 为虚拟页建立页目录项和页表项, 使虚拟页内的地址经过分页机制变换到物理页内。

```
if (physicalPageAddress)
{
    //printf("allocate physical page 0x%x\n",
    physicalPageAddress);
}
```

```

        // 第三步：为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理页内。
        flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
    }
    else
    {
        flag = false;
    }

    // 分配失败，释放前面已经分配的虚拟页和物理页表
    if (!flag)
    {
        // 前 i 个页表已经指定了物理页
        releasePages(type, virtualAddress, i);
        // 剩余的页表未指定物理页
        releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
        return 0;
    }
}

```

进入函数 connectPhysicalVirtualPage。

```

// 第三步：为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理页内。
flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);

```

```

bool MemoryManager::connectPhysicalVirtualPage(const int virtualAddress, const int physicalPageAddress)
{
    // 计算虚拟地址对应的页目录项和页表项
    int *pde = (int *)toPDE(virtualAddress);
    int *pte = (int *)toPTE(virtualAddress);

    // 页目录项无对应的页表，先分配一个页表
    if (!(*pde & 0x00000001))
    {
        // 从内核物理地址空间中分配一个页表
        int page = allocatePhysicalPages(AddressPoolType::KERNEL, 1);
        if (!page)
            return false;

        // 使页目录项指向页表
        *pde = page | 0x7;
        // 初始化页表
        char *pagePtr = (char *)(((int)pte) & 0xfffff000);
        memset(pagePtr, 0, PAGE_SIZE);
    }

    // 使页表项指向物理页
    *pte = physicalPageAddress | 0x7;

    return true;
}

```

```
}
```

第一步是计算虚拟地址对应的页目录项。利用 toPDE 函数得到了 0xc010000 对应的页目录项地址为 0xfffffc00。从该地址取值为 0x00101067，这个应该是页目录项的值。

```
int MemoryManager::toPDE(const int virtualAddress)
{
    return (0xffff000 + (((virtualAddress & 0xffc00000) >> 22) * 4));
}
```

```
(gdb) p/x pde
$8 = 0xfffffc00
```

```
(gdb) x/wx 0xfffffc00
0xfffffc00: 0x00101067
```

第二步是计算虚拟地址对应的页表项。利用 toPTE 函数得到了 0xc010000 对应的页表项地址为 0xffff00400。

```
int MemoryManager::toPTE(const int virtualAddress)
{
    return (0xffc00000 + ((virtualAddress & 0xffc00000) >> 10) + (((virtualAddress
```

```
(gdb) p/x pte
$9 = 0xffff00400
```

不知道为什么要让 0x200000 和 0x7 做异或，实现 U/S = 1, R/W = 1, P = 1，最后得到的页表项的值为 0x00200007。

```
// 使页表项指向物理页
*pte = physicalPageAddress | 0x7;
```

```
(gdb) x/wx 0xffff00400
0xffff00400: 0x00200007
```

最后返回原函数。可以看到 flag 已经便成了 true，意味着虚拟地址 0xc0100000 和物理地址 0x200000 建立了联系。综上我们可以知道，0xc0100000 的页目录项 0x00101067，页表项 0x00200007。

```
// 第三步：为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理地址
flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
```

```
(gdb) p flag
$13 = true
```

到此意味着 p1 已经被成功分配。下面进行页内存的释放。

手动释放 p2 的 10 页页内存。

```
memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
```

释放页内存的代码如下。这个代码是对顺序连续的虚拟地址的不连续物理地址逐页释放，最后统一释放虚拟内存。

```
void MemoryManager::releasePages(enum AddressPoolType type, const int virtualAddress, const int count)
{
    int vaddr = virtualAddress;
    int *pte, *pde;
    bool flag;
    const int ENTRY_NUM = PAGE_SIZE / sizeof(int);

    for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
    {
        releasePhysicalPages(type, vaddr2paddr(vaddr), 1);

        // 设置页表项为不存在，防止释放后被再次使用
        pte = (int *)toPTE(vaddr);
        *pte = 0;
    }

    releaseVirtualPages(type, virtualAddress, count);
}
```

p2 的首地址如下。

```
(gdb) p/x vaddr
$16 = 0xc0164000
```

对每一个虚拟页释放为其分配的物理页。

```
for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
{
    // 第一步，对每一个虚拟页，释放为其分配的物理页
    releasePhysicalPages(type, vaddr2paddr(vaddr), 1);
}
```

从虚拟地址求物理地址。返回物理地址如下。

```
int MemoryManager::vaddr2paddr(int vaddr)
{
    int *pte = (int *)toPTE(vaddr);
    int page = (*pte) & 0xfffff000;
    int offset = vaddr & 0xfff;
    return (page + offset);
}
```

```
(gdb) p/x page+offset
$18 = 0x264000
```

将 pte 置为 0，设页表项不存在。

```
// 设置页表项为不存在，防止释放后被再次使用
pte = (int *)toPTE(vaddr);
*pte = 0;
```

最后释放虚拟页。

```
// 第二步，释放虚拟页
releaseVirtualPages(type, virtualAddress, count);
```

显示释放了 10 页后的 p2 首地址如下。

```
(gdb) p/x p2
$21 = 0xc01d2000
```

2.构造测试例子来分析虚拟页内存管理的实现是否存在 bug。如果存在，则尝试修复并再次测试。否则，结合测例简要分析虚拟页内存管理的实现的正确性。

在 assignment 1 中发现当一个分配内存全部释放后，原来的指针仍然指向释放前的首地址。不知道这个算不算 bug。



bug 展示如下：

首先在实现任务 3 的基础上，在 setup 的 first\_thread 中增加一个对 p7 的完全释放/部分释放。

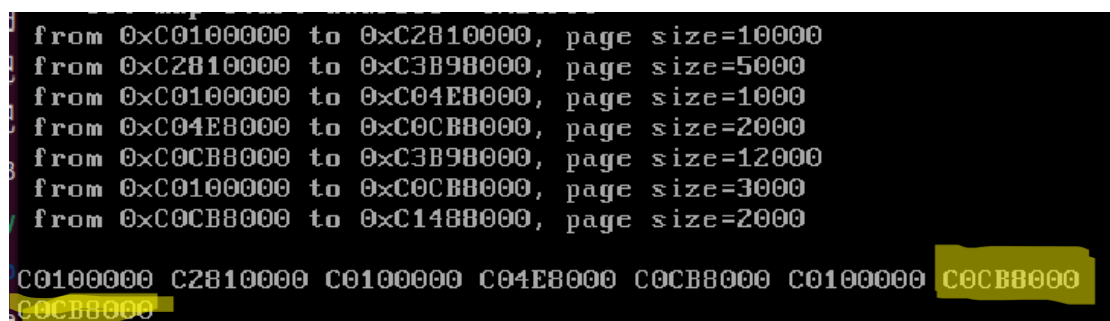
```
void first_thread(void *arg)
{
    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10000);
    char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 5000);
    char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1000); //从头分
    char *p4 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 2000);
    char *p5 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 12000);
    char *p6 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 3000); //从头分
    char *p7 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 2000);

    printf("\n%x %x %x %x %x %x %x \n", p1, p2, p3, p4, p5, p6, p7);

    memoryManager.releasePages(AddressPoolType::KERNEL, (int)p7, 2000);
    printf("%x", p7);

    asm_halt();
}
```

输出如下。



```
from 0xC0100000 to 0xC2810000, page size=10000
from 0xC2810000 to 0xC3B98000, page size=5000
from 0xC0100000 to 0xC04E8000, page size=1000
from 0xC04E8000 to 0xC0CB8000, page size=2000
from 0xC0CB8000 to 0xC3B98000, page size=12000
from 0xC0100000 to 0xC0CB8000, page size=3000
from 0xC0CB8000 to 0xC1488000, page size=2000

C0100000 C2810000 C0100000 C04E8000 C0CB8000 C0100000 C0CB8000
C0CB8000
```

可以发现不管是部分释放、还是完全释放后输出 p7 居然还是原来的首地址。按理来说不应该是这样的。releasePages 的逻辑是从虚拟地址的首地址开始连续释放掉前 count 页虚拟地址的物理地址，随后统一释放虚拟地址。如果是部分释放，释放过后虚拟地址的首地址应该是原来的首地址加上 count\*4096；如果是部分释放，释放过后首地址应该都不存在了。

以下做出修改。

修改 releasePages。首先把 releasePages 函数的返回值改成 int 型，返回释放内存后的首地址。增加参数 beforeSize（记录原来分配的内存大小），仅用来和 count 做判断。当 beforeSize 和 count 大小一致，意味着分配的内存完全释放，此

时返回-1 作为首地址。当 beforeSize 大于 count，意味着分配的内存部分释放，此时仅需要将传入的首地址加上 count\*4096，返回的就是更新后的首地址。

```
int MemoryManager::releasePages(enum AddressPoolType type, const int virtualAddress, const int beforeSize, const int count)
{
    int vaddr = virtualAddress;
    int *pte;
    for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
    {
        // 第一步，对每一个虚拟页，释放为其分配的物理页
        releasePhysicalPages(type, vaddr2paddr(vaddr), 1);

        // 设置页表项为不存在，防止释放后被再次使用
        pte = (int *)toPTE(vaddr);
        *pte = 0;
    }

    // 第二步，释放虚拟页
    releaseVirtualPages(type, virtualAddress, count);

    if(beforeSize==count) return -1;

    return vaddr+count*4096;
}
```

当对 p7 仅释放 1000 页：

```
p7 = (char*)memoryManager.releasePages(AddressPoolType::KERNEL, (int)p7, 2000,1000);
printf("%x",p7);
```

可以发现释放后的首地址已经更新。首地址后移了 1000 页。

```
from 0xC0100000 to 0xC2810000, page size=10000
from 0xC2810000 to 0xC3B98000, page size=5000
from 0xC0100000 to 0xC04E8000, page size=1000
from 0xC04E8000 to 0xC0CB8000, page size=2000
from 0xC0CB8000 to 0xC3B98000, page size=12000
from 0xC0100000 to 0xC0CB8000, page size=3000
from 0xC0CB8000 to 0xC1488000, page size=2000

C0100000 C2810000 C0100000 C04E8000 C0CB8000 C0100000 C0CB8000
C1488000
```

当对 p7 完全释放（一共 2000 页）：

```
p7 = (char*)memoryManager.releasePages(AddressPoolType::KERNEL, (int)p7, 2000,2000);
printf("%x",p7);
```

可以观察到首地址直接变成了-1。

```
from 0xC0100000 to 0xC2810000, page size=10000
from 0xC2810000 to 0xC3B98000, page size=5000
from 0xC0100000 to 0xC04E8000, page size=1000
from 0xC04E8000 to 0xC0CB8000, page size=2000
from 0xC0CB8000 to 0xC3B98000, page size=12000
from 0xC0100000 to 0xC0CB8000, page size=3000
from 0xC0CB8000 to 0xC1488000, page size=2000

C0100000 C2810000 C0100000 C04E8000 C0CB8000 C0100000 C0CB8000
```

到这里这个 bug 算修改完成！

## Section 5 实验总结与心得体会

这一次实验学习到了很多东西。首先是阅读非常长的 readme 学习到了更多的知识，才发现实验起来很多细节和原来学的理论知识不太一样。就那个映射关系我到现在还没有啃透……设计最优分配算法时还好，设计 fifo 页面置换算法就遇到了很多困难。首先是基础不牢在链表的实现上吃了亏，然后在一处计算中少释放了内存导致分配错误，卡了很久。总之很高兴都解决了，成就感还是满满的。学习到了很多东西，感谢把知识点写的那么浅显易懂的助教 and 老师。